Parveen Khurana

124 Followers About Following

Sequence Learning Problems

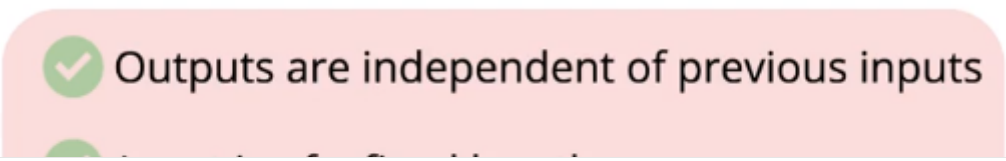
P Parveen Khurana Jul 6, 2019 · 15 min read

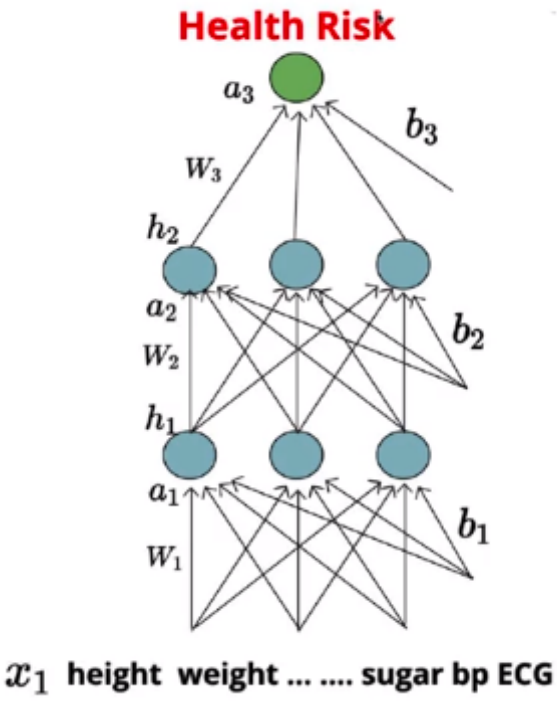
This article covers the content discussed in the Sequence Learning Problems module of the Deep Learning course and all the images are taken from the same module.

In all of the networks that we have covered so far(Fully Connected Neural Network(FCNN), Convolutional Neural Network(CNN)), **the output at any time step is independent of the previous layer input/output and the input was always of the fixed-length/size** for ex. for FCNN all the input instances had the same let's say '**100**' input features whereas in case of CNN's let's say all the input images are of size '**30 X 30**' or if of different size, then we can rescale the input image to the required/appropriate dimension. And the other property of these architectures/networks is that all the neurons in any of the layers are connected to all the neurons in the previous layer(For CNNs, we can think of it as the weights associated with most of the neurons from the previous layers is 0 and we consider only a few neurons).

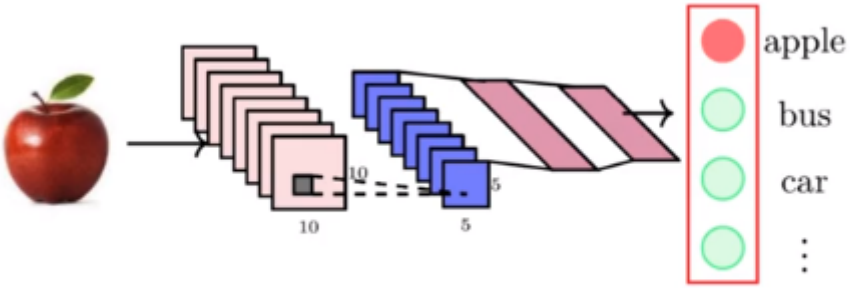
Two properties of FCNN and CNN:

- Output at any time step is independent of previous inputs
- Input is of a fixed length





Fully Connected Neural Network



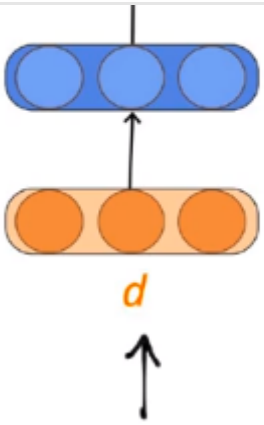
Convolutional Neural Network

Sequence Learning Problems

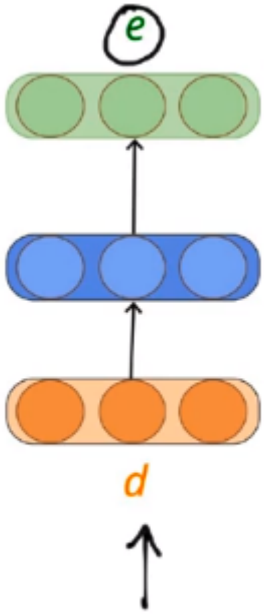
In Sequence Learning Problems, the two properties of FCNN and CNNs do not hold and the output here at any timestep depends on previous input/output and the length of the input is not fixed.

Let's consider the case of Auto-completion. We type in the alphabet '**d**', then it tries to predict the next character.

e



See this as the classification problem, the job is to identify the next character/alphabet of the 26 alphabets given that you have started typing the alphabet ‘d’. We can treat this as the output is a distribution over these 26 characters.



There is some true distribution and in this case, all the probability mass is on ‘**e**’(assuming it is the true character that should come after ‘**d**’) and the probability mass would be 0 for all other characters in the true distribution i.e **y = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]**.

Our predicted output **y_hat** is also going to be a distribution over these 26 characters. The output layer here as well would be a **softmax layer**.

‘**a**’ would be [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

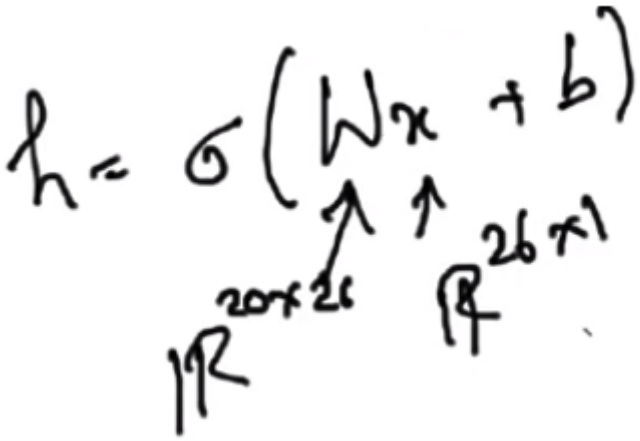
‘**b**’ would be [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

i.e **each character/alphabet is represented by a 26-dimensional vector**(where the numbers in the array is the total number of possible values that the variable can take) and the element in this vector at the index corresponding to the index of the alphabet(index starts from 0) would be set to 1 and everything else would be 0.

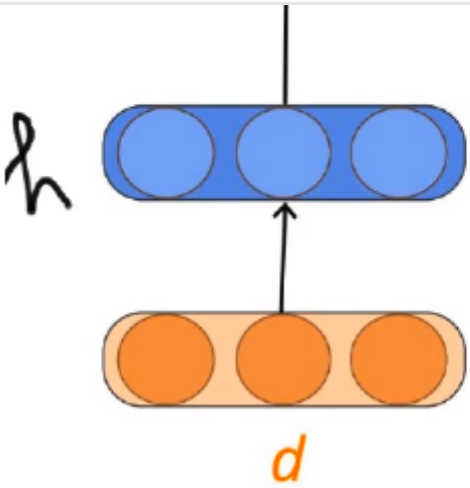
So, the input ‘**d**’ is represented as [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

In this case, input and output have the same dimension but that may not be the case always.

Now we have 1 hidden layer and then the final output layer. Let’s say the **hidden layer dimension is ‘20 X 1’** and **the input and output dimension is ‘26 X 1’**. Then the hidden layer would be computed as:



And similarly, bias(**b**) in the above equation is going to be a 20-dimensional vector, so (**Wx + b**) would give us a 20-dimensional vector and then we apply non-linearity on this and we get the output at this intermediate layer as ‘**h**’(this represents the value at the hidden layer, not to be this confused with alphabet ‘h’) which is going to be a 20-dimensional vector.



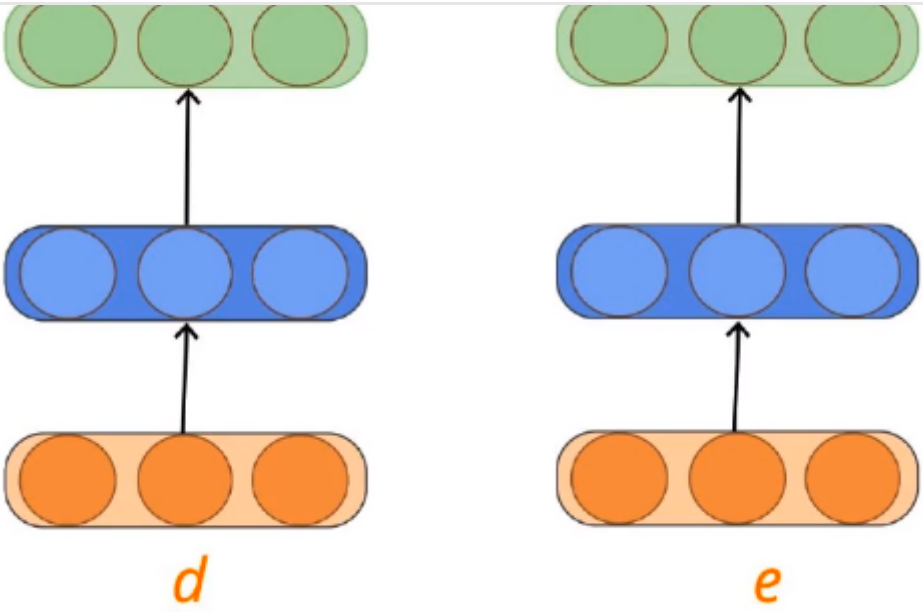
$$h = \sigma(Wx + b)$$

$$\hat{y} = \text{softmax}(\underbrace{Vh + c}_{26 \times 20 + C})$$

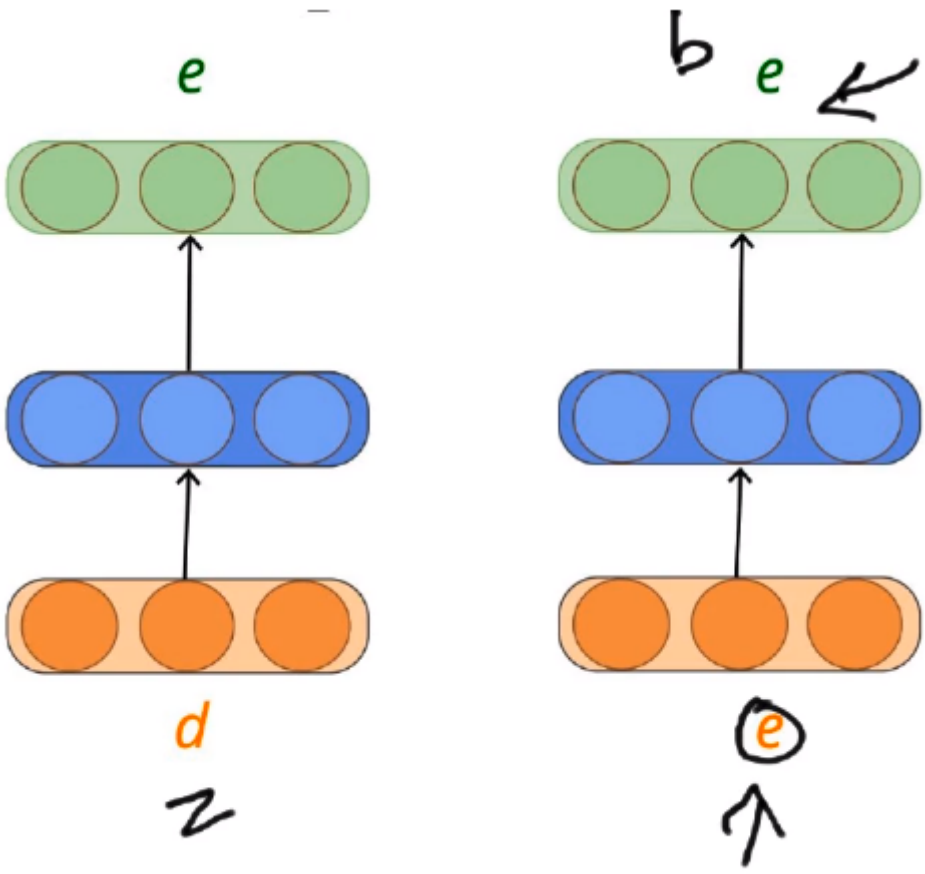
As '**h**' is 20 dimensional and we want the output to be 26 dimensional(over the 26 alphabets), so '**V**' would be of size (26 X 20) and '**c**' would be 26-dimensional vector, we do the arithmetic as (**Vh + c**) and apply the softmax on the output of this arithmetic and that would give the value of '**y_hat**' as the probability distribution.

The way Auto-completion works is that it takes the top 3–4 entries from this '**y_hat**' distribution and it suggests those characters.

Let's say out of these top 3, '**e**' was there and user-selected '**e**' as the second character(after '**d**')

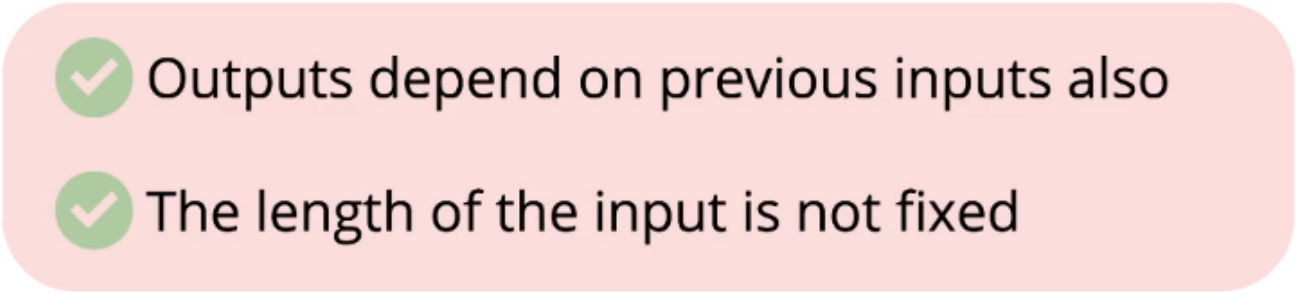


Now this '**e**' acts as the input and the model again tries to predict an output. The interesting part is that the output in this case now not only depends on the input('**e**'), it also depends on the input in the previous layer:



second layer is going to be '**b**' to complete word '**zebra**' instead of '**e**'.

So, the outputs in Sequence Learning Problems depend on previous input also and the length of the input is not fixed. We have a sequence of inputs and the output depends on all previous sequences of inputs or at least some sequence of input for example in sentence completion, the 100th word might not depend on all the previous 99 words but for sure it would depend on the last 3–4 words.



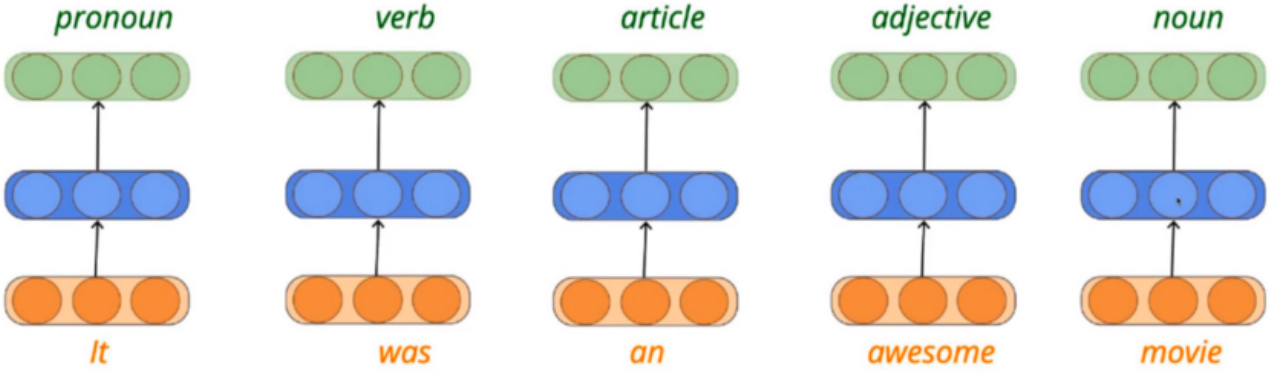
The length of the input is also not fixed, for example for the above example we had 4 characters as the input but if we take a longer or a shorted word, then the number of characters would change and that is what we mean by the length of the input.

And instead of having the distribution over 26 characters, we could have it over say (26 + 1) characters where this additional character tells us that the sequence has ended.

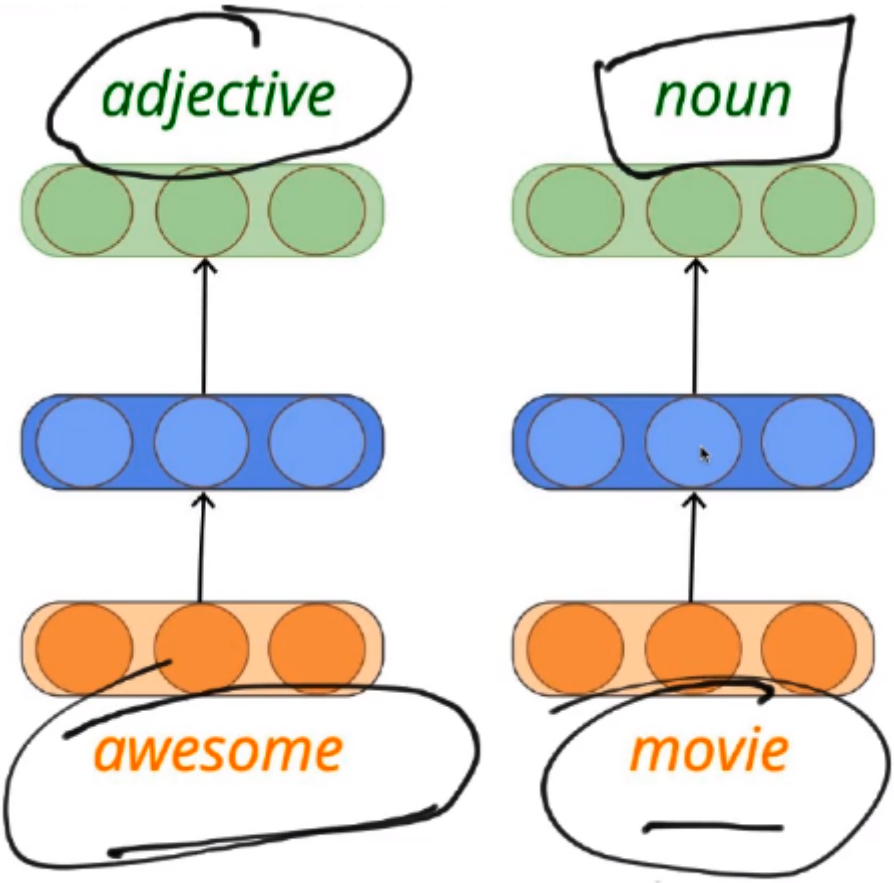
Some more examples of Sequence Learning Problems

Part of speech tagging:

We are given a sequence of words and for each word we want to predict part of the speech tag of that word(means predicting whether that word is a pronoun, noun, verb, article, adjective and so on).



previous input was '**awesome**' which is an adjective; the moment we see an adjective we're more or less confident that the next word is actually going to be a noun. So, the confidence in predicting that the '**movie**' is a noun would be higher if we know that the previous input was an adjective. So, there is this dependency where the current output depends not only on the current input but on the previous input as well.



Let' say we have the word '**bank**' in a sentence. Now, this could be a Verb(I can bank on him) or a Noun(I had gone to **a** bank). In the Noun case, we can see that the previous word is an article, from which we get to know that the following word is very unlikely to be a verb, it is going to be a Noun. So, even in this type of ambiguous cases, we look at the previous sequence of words(the context) and we are in a better position to make this decision.

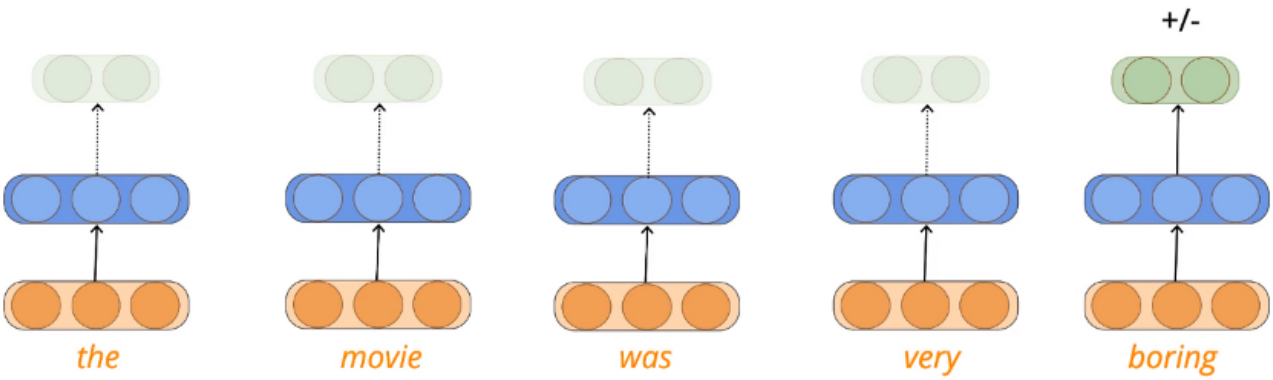
Here also, we use the same one hot encoding technique to represent words in form of numbers. Just list down all the words and assign them an index and stick to that index and when we encounter a word, look at its corresponding index, let's say it's

this input, we compute the '**wx + b**' value, apply non-linearity on it and pass to subsequent layers and finally through the softmax layer which gives the probability distribution.

So, here also the output depends on previous inputs and the length of the input is not fixed(sentence could be of any number of words).

Sentimental Analysis:

We look at all the words in a sentence and give a final output as positive/negative sentiment, we don't produce output for each word(time step) instead we take into consideration all the words and from that, we predict only 1 output at the end. We can think of it like we are producing the output at every time step but we are ignoring those output and reporting only the final output and somehow we need to make this final output dependent on all the previous inputs as well. This is also termed as Sequence Classification Problem.

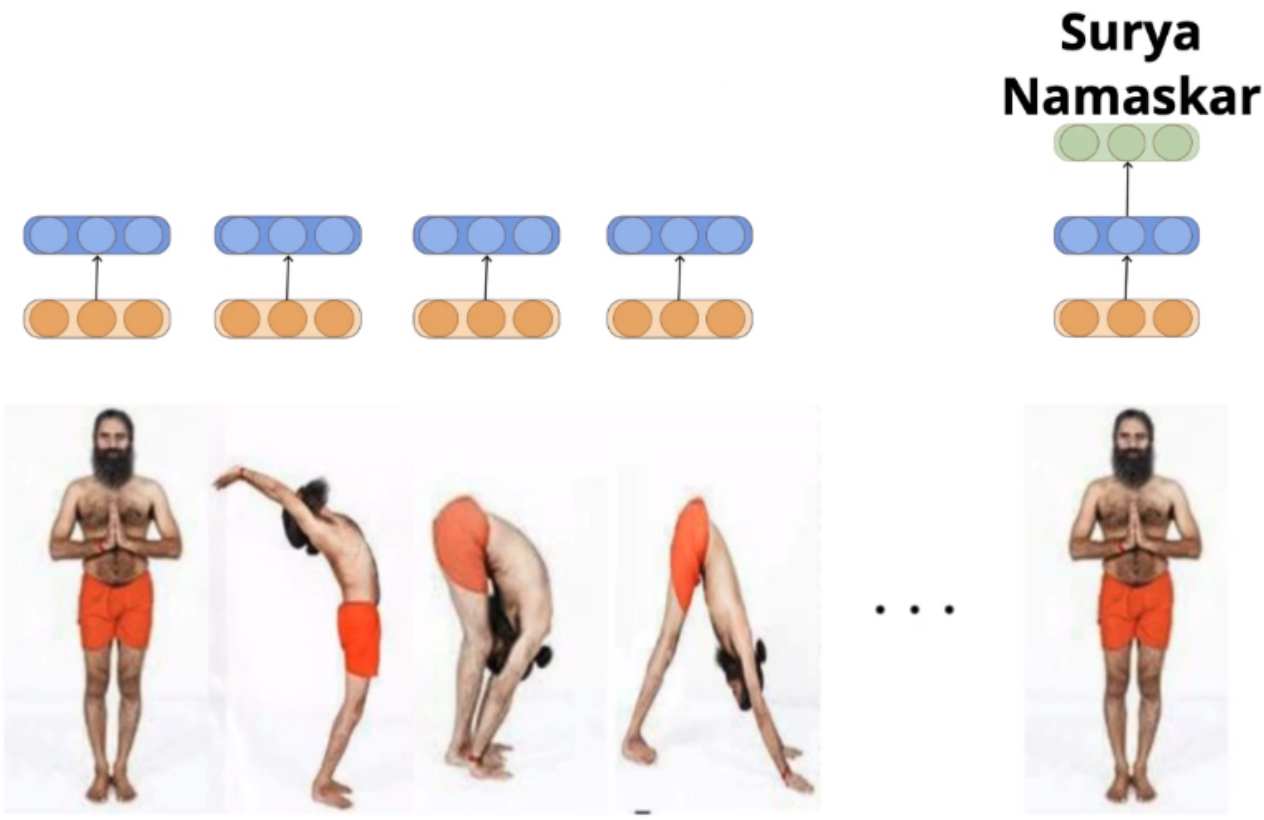


Other Kind of Problems:

Speech Recognition —We can think of speech as a sequence of phonemes and we take the speech signal as the input and tries to map to phonemes in a language.

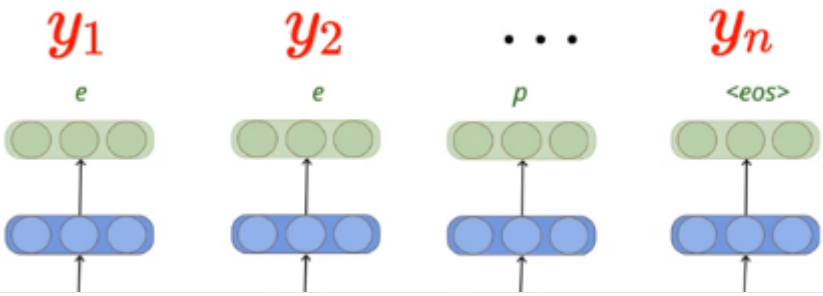
Another thing we can do is to look at the entire sequence of speech than say whether the person is speaking angrily, is happy/sad and etc.

Video Labeling — A video is a sequence of frames and we can do some processing on these frames, we can do labeling of every frame in the video; we can look at all the frames and give the label for the entire video. Input is again variable in this case as the no. of frames would be different for different videos. At every time step, we are taking in one frame as the input and in the end, we are assigning a label to the video.



How to model sequence learning problems?

Let's consider the character prediction program(in which we predict output at every time step):



$$x_1 \qquad x_2 \qquad \dots \qquad x_n$$

$$y_t = \hat{f}(x_1, x_2, \dots, x_t)$$

We call the output at the 1st-time step as $\mathbf{y_1}$, output at the 2nd-time step as $\mathbf{y_2}$ and so on. So, in general, we are calling the output at the t'th time step as $\mathbf{y_t}$(where '**t**' is the time step).

We know that the true relationship is of such form that $\mathbf{y_t}$ depends on all the input that we have seen so far

$$y_t = f(x_1, x_2 \dots x_t)$$

True relation

It may not depend on all the previous inputs but at least it depends on some of the previous inputs apart from the current inputs.

We need some function that takes into account previous inputs to predict **y_hat**. In other words, we need our function/model to approximate the relation between the input and the output and ensure that the function somehow depends on the previous input as well.

$$y_t = \hat{f}(x_1, x_2 \dots x_t)$$

Approximation of the true relationship

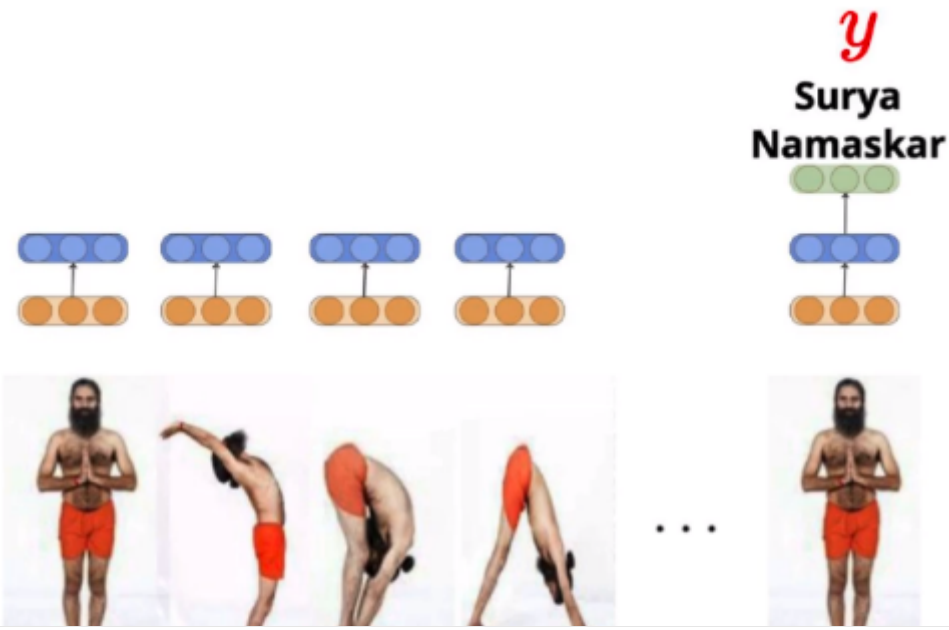
At every time step, we want to do the same processing, we take the current input, the history of the inputs and then produce some output and the output is also from the same set(from 26 alphabets in this case). So, the task is not changing from one time step to the other time step, the only thing changing is the input to the function.

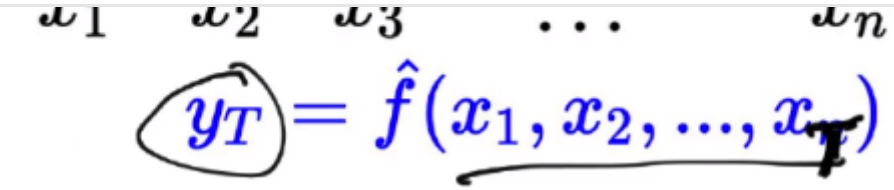
The function should be able to work with the variable number of inputs -> let say for this word there are 4 characters, for some other word there could be 10 characters, so the function should be able to take in a variable number of inputs.

We want a function which can act as a good approximation of the relation between the input and the output while serving these 3 things:

- The same function should be executed at every time step
- The function should consider the current as well as previous inputs
- The function should be able to deal with a variable number of inputs

For the below case we are predicting just one output from all the inputs, so here as well the same conditions should hold i.e the function should be able to deal with a variable number of inputs and the output should depend on all the previous inputs. Here the other point(same function being executed at every time step does not hold) because we are not making a prediction at every time step





- Ensure that y_t is dependent on previous inputs also
- Ensure that the function can deal with variable number of inputs
- Ensure that the function executed at each time step is the same

Let's start with the requirement "the function being executed at each time step is the same" and the rationale behind this requirement was that at every time step, the function is doing the same job, it takes an input, a history of inputs and predict the output(say the part of the speech tag)

$$h_i = \sigma(W_1 x_i + b_1)$$
$$y_i = O(W_2 h_i + b_2)$$
$$i = timestep$$

Let the input at the i'th time step is: $\mathbf{x_i}$

We multiply the input with the weight matrix, add bias to it, apply the non-linearity on the summation of $\mathbf{W_1 x_i}$ and $\mathbf{b_1}$ and this gives us the $\mathbf{h_i}$ for the i'th input.

Then we repeat the same thing i.e we multiply $\mathbf{h_i}$ with weight matrix, add bias to it, apply non-linearity and compute the output(we take the dimensions in such a way that the output has the probability distribution over all the classes that we have)

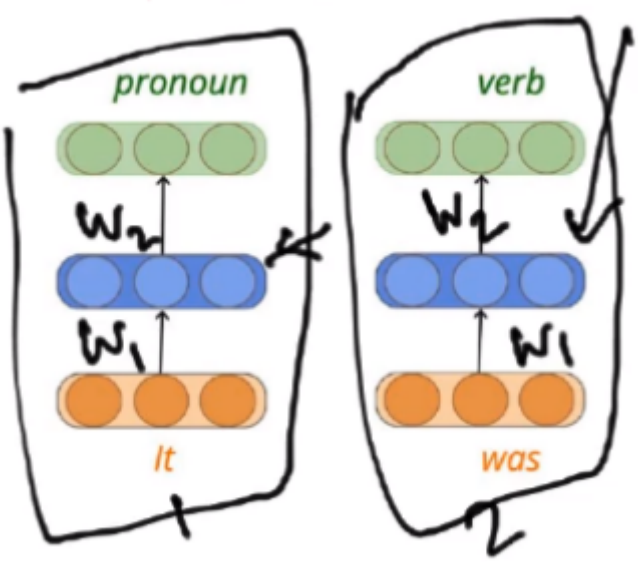
$$y_i = O(W_2 h_i + b_2)$$

$$i = timestep$$

$$y_1 = O(W_2 h_1 + b_2)$$

$$y_2 = O(W_2 h_2 + b_2)$$

The network parameters i.e **W1, W2, b1, b2** remain the same(at every time step) and hence the function is also the same. So, this is achieved using Parameter Sharing(same parameters at all the time steps).



In RNNs the nomenclature is a little different compared to FCNN

$$s_i = \sigma(U x_i + b)$$

$$y_i = O(V s_i + c)$$

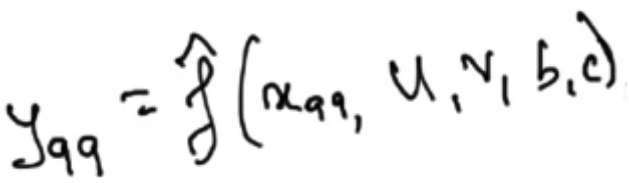
b1, **b2** are represented by **b**, **c**

$\mathbf{h_i}$ is represented by $\mathbf{s_i}$

s is termed as the state

With the above function, the 2nd property is also achieved i.e the above-mentioned function can deal with a variable number of inputs.

Let's say we want to compute **y99**, then we have

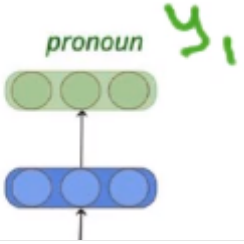


So, we have chosen the function in a way that it is irrespective of the time steps, even if there are variable number of inputs, we just need to plug in the current time step's input into the formula and get the output but the downside of this is that we have not ensured the first most property that $\mathbf{y_t}$ is dependent on previous inputs also.

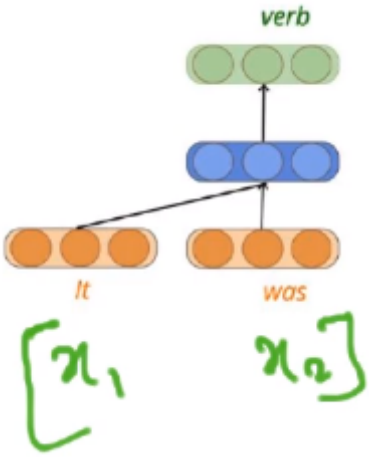
We want our function to be something like:

$$y_1 = f(x_1)$$
$$y_2 = f(x_1, x_2)$$
$$y_3 = f(x_1, x_2, x_3)$$

At the first time step, the output depends only on the first input.



Then at the second time step, we can concatenate the first two inputs and pass this concatenated input through the network to compute the output



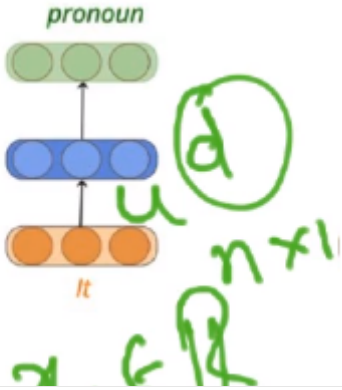
Here we concatenate **x1, x2** into **x** and multiply **x** with **U**, then we calculate $\mathbf{y_i}$ using the value of $\mathbf{s_i}$.

Now the situation is that **y1** is a function of **x1**

y2 is a function of **x1, x2**

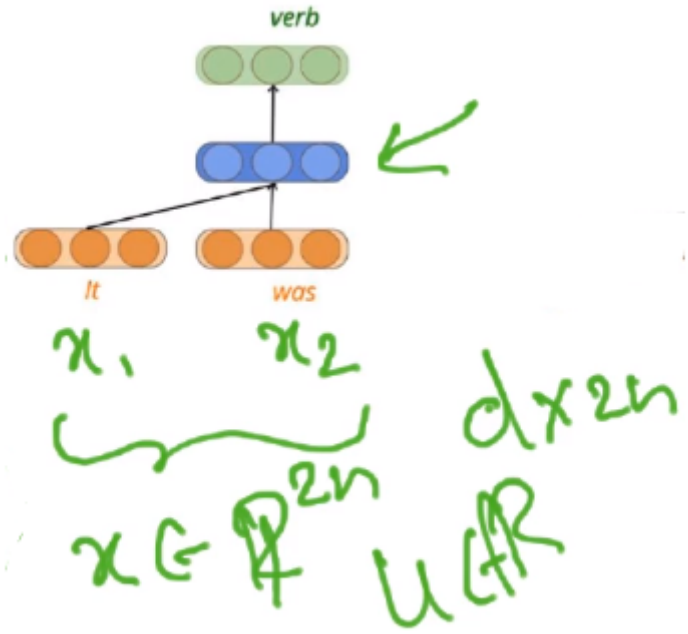
for **y3**, we concatenate **x1, x2, x3** and call it as **x**, pass this **x** and compute $\mathbf{s_i}$ and from $\mathbf{s_i}$ we compute $\mathbf{y_i}$ so here **y3** would be a function of **x1, x2, x3**

If we follow this approach, then the function being executed at every time step is not the same and as a result of that, parameter sharing won't be there:



At the first time step, only **x1** is there, that would be a vector of dimension (**nX1**), we compute the hidden representation(‘**d**’ dimensional) using **x1**, then the dimension of **u** would be (**d X n**)

Now at the second time step, we concatenate **x1** and **x2** and call it as **x**, so the dimension of **x** here would be (**2n X 1**) and the dimension of **u** would be (**d X 2n**).

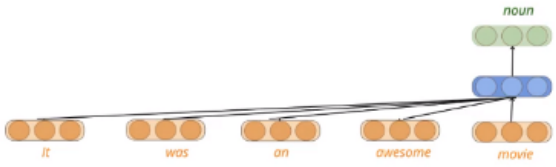


So, the ‘**u**’ that we use in the second time step is going to be different than the ‘**u**’ used in the first time step -> parameters not being shared => function being executed at every step is going to be different.

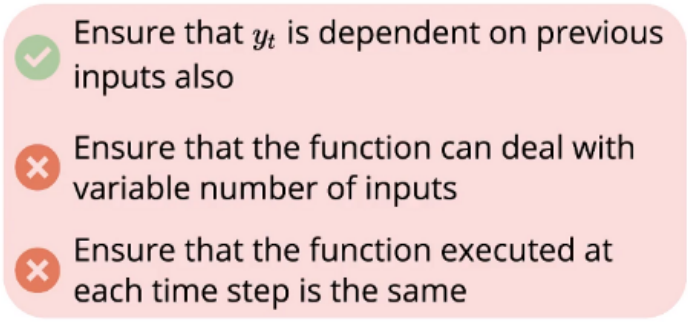
If we use addition instead of concatenation, in that case, dimension would remain the same but there is a catch here that at every time step, the semantics of the input is changing; at the first time step the input is just **x1**; at second time step it is (**x1 + x2**) so the semantics is changing; then we have (**x1 + x2 + x3**) so again the meaning of the input actually changes, it's not the same as just feeding 1 word, now we have created

This is not serving the purpose:

- At every time step, it is increasing the number of parameters
- The function being executed at every time step is different
- Now we can't deal with arbitrary input size because let's say at training time all our data was of length up to 20 only but at test time if we get sentence of length 21, now it needs a new u parameter of dimension (d X 21n), and this parameter was not in the training.



$$y_n = f(x_1, x_2, x_3, ..., x_n)$$

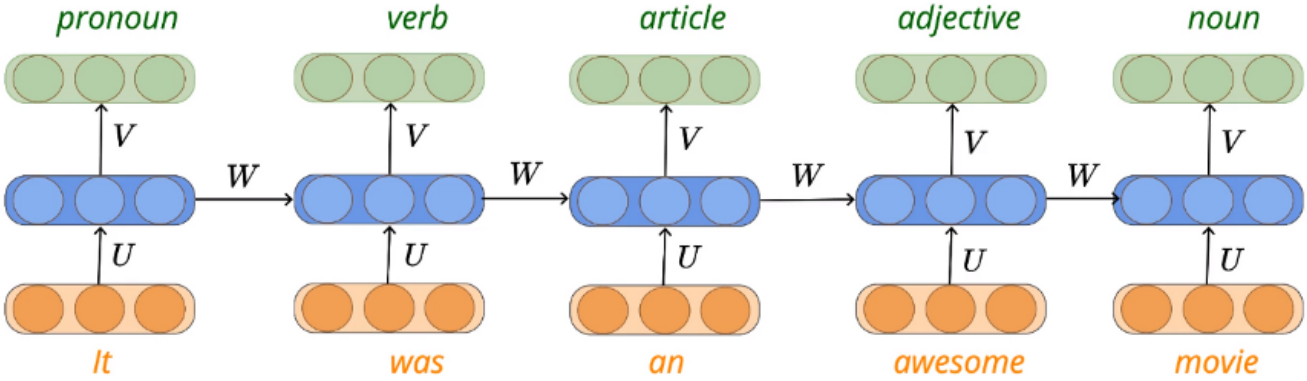


RNNs

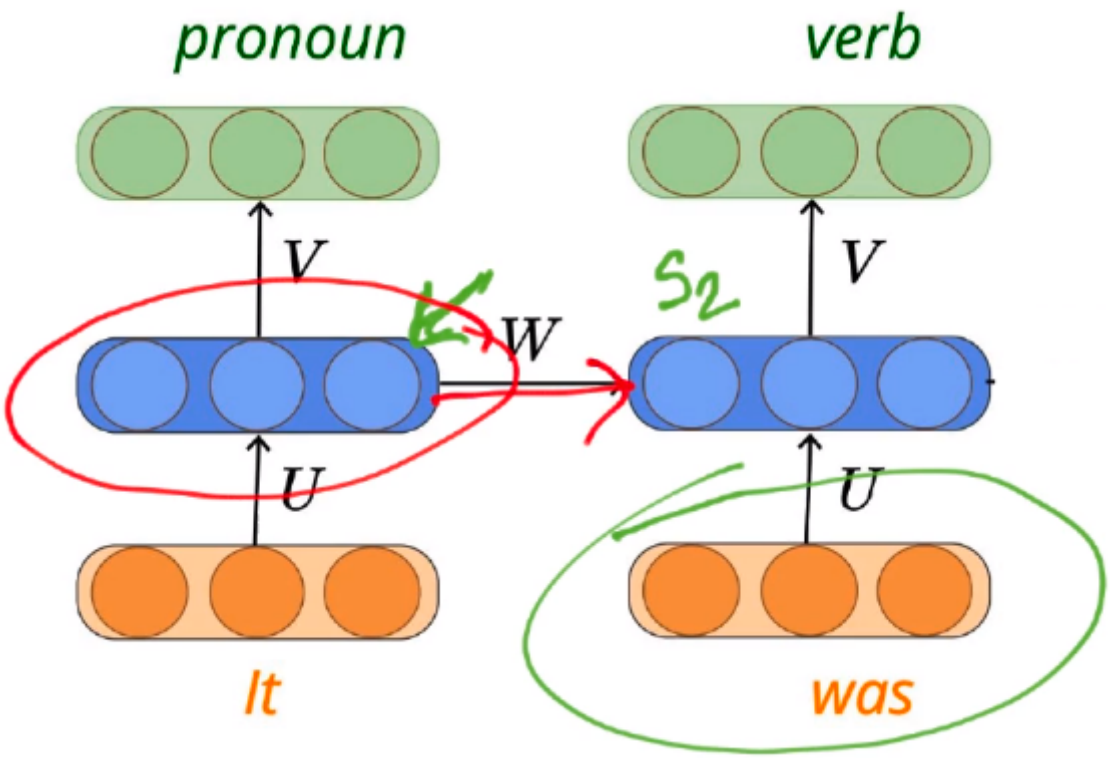
To make sure that all 3 properties are satisfied, we re-define the model's equation as:

$$s_i = \sigma(Ux_i + Ws_{i-1} + b)$$

$$y_i = O(Vs_i + c)$$



Let's say we want to compute **s2**, **s2** depends on **x2** and **s1** and **s1**, in turn, depends on **x1**, so ultimately the input at the 2nd time step also depends upon the input for the first time step:



Dimensions of different parameters:

U -> d X n

xi -> n X 1

W -> d X d

s(i-1) -> d X 1

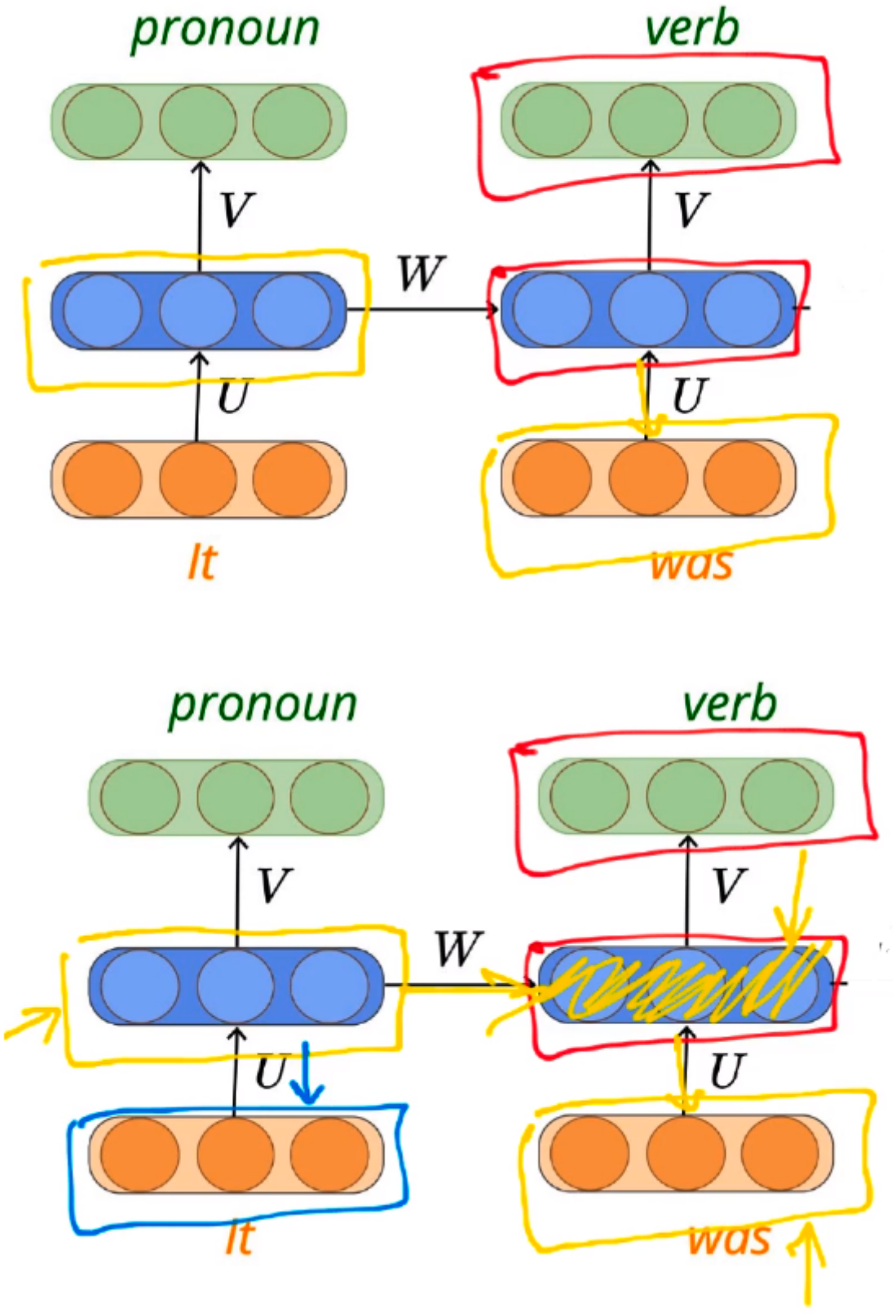
b -> d X 1

si -> d X 1

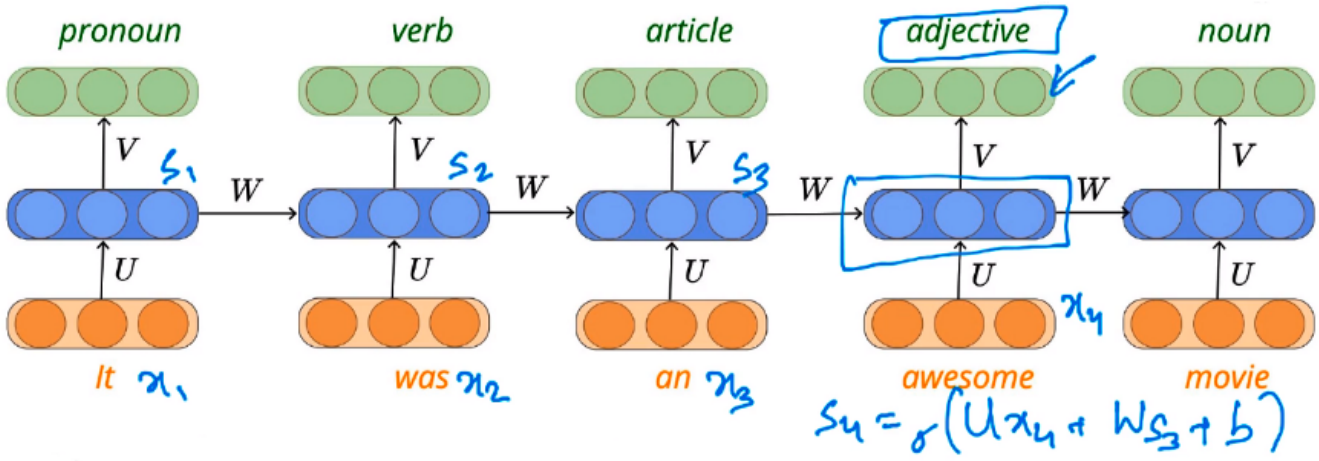
V -> k X d (assuming there are k output classes)

So, with this new formula, it is clear that the same function is going to be executed at all time steps, all the parameters remain the same only the input changes.

depicted in the image below:



$$s_i = \sigma(Ux_i + Ws_{i-1} + b)$$



And this function can deal with a variable number of inputs as well.

$$y_i = \hat{f}(x_i, s_{i-1}, W, U, V, b, c)$$

So, this is what a Recurrent Network looks like, we have these recurrent connections between the hidden states that ensures all the requirements that we have for the model are met.

Rnn Recurrent Neural Network Sequence Learning Deep Learning Artificial Neural Network